Schema-guided Testing of Message-Oriented Systems

André Santos[1,2][a], Alcino Cunha[2][b] and Nuno Macedo[3][c]
[1]*VORTEX CoLab, Vila Nova de Gaia, Portugal*
[2]*High-Assurance Software Laboratory, INESC TEC and University of Minho, Braga, Portugal*
[3]*High-Assurance Software Laboratory, INESC TEC and University of Porto, Porto, Portugal*
andre.santos@vortex-colab.com, alcino@di.uminho.pt, nmacedo@fe.up.pt

Keywords: Software Testing, Formal Specifications, Specification-based Testing, Property-based Testing.

Abstract: Effective testing of message-oriented software requires describing the expected behaviour of the system and the causality relations between messages. This is often achieved with formal specifications based on temporal logics that require both first-order and metric temporal constructs – to specify constraints over data and real time. This paper proposes a technique to automatically generate tests for metric first-order temporal specifications that match well-understood specification patterns. Our approach takes in properties in a high-level specification language and identifies test schemas (strategies) that are likely to falsify the property. Schemas correspond to abstract classes of execution traces, that can be refined by introducing assumptions about the system. At the low level, concrete traces are successively produced for each schema using property-based testing principles. We instantiate this approach for a popular robotic middleware, ROS, and evaluate it on two systems, showing that schema-based test generation is effective for message-oriented software.

1 INTRODUCTION

As the complexity of software systems increases, so does the necessity to properly verify that safe behaviour will be observed. Testing is an essential, and often the only, strategy deployed for that purpose, but the complexity of modern systems renders the manual encoding of test cases impractical. As a consequence, several automated approaches to test generation have emerged, such as model-based (MBT), property-based (PBT), and specification-based testing (SBT).

These challenges are exacerbated when targeting distributed systems where components communicate asynchronously through message-passing, such as those complying to the OMG's DDS standard (OMG, 2015). These systems are typically heterogeneous and built from third-party components, so system-level testing should act at the message-passing level, treating components as black-boxes. In this context, the behaviour to be analysed is of a dynamic nature, and testing procedures must take into consideration full traces of messages as inputs and outputs. This limits the feasibility of using unit tests beyond specific corner cases. PBT approaches are also ill-suited since they either focus on execution safety properties (such as null references or buffer overflows) or require the expected behaviour to be specified operationally. MBT approaches require modelling the system behaviour, which is often infeasible in systems of this nature.

SBT approaches can be used to tackle these issues. Here, the user is only expected to provide a single artefact, a high-level declarative specification of the expected behaviour of the system. In a dynamic context, these take the shape of temporal specifications, allowing the encoding of functional safety properties. Relevant trace inputs are automatically generated by inspecting the specifications, while validity is automatically tested by evaluating the specification against the output traces. However, previously proposed techniques of this nature are still affected by some issues. First, they are often based on logics with abstract time, such as linear temporal logic (LTL) (Tan et al., 2004; Michlmayr et al., 2006; Arcaini et al., 2013; Bloem et al., 2019; Narizzano et al., 2020), but most systems are expected to be tested for some timing constraints. Moreover, they often fail to provide a proper high-level interface that spares the developer from having to understand the underlying formalisms. Lastly, due to the extent of the search-space for timed trace test cases, such techniques must provide some kind

[a] https://orcid.org/0000-0002-1985-8264
[b] https://orcid.org/0000-0002-2714-8027
[c] https://orcid.org/0000-0002-4817-948X

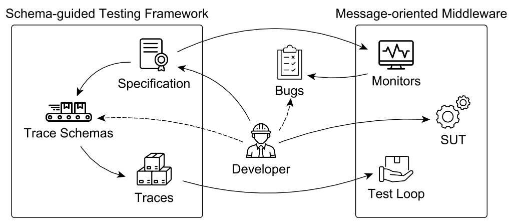

Figure 1: Overview of the proposed approach.

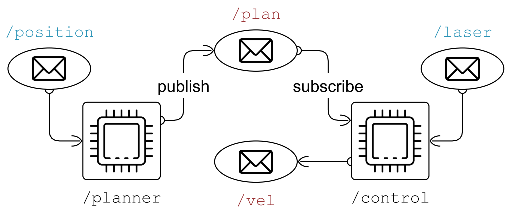

Figure 2: ROS system with open subscribers.

of mechanisms to guide the input generation. Some techniques have been proposed in PBT, such as target-oriented (Löscher and Sagonas, 2017) and coverage-guided (Padhye et al., 2019) approaches, but they are ill-suited for heterogeneous and distributed systems.

In this paper we propose a novel SBT approach, depicted in Figure 1, based on *trace schemas* for message-oriented systems whose expected behaviour is specified in a high-level specification language. The users are only required to specify the architecture and the expected behaviour of the system under test (SUT), from which the input generator and the property monitors are derived. The language is based on well-known specification patterns (Dwyer et al., 1999), so that knowledge about the underlying formalisms or the implementation details of the SUT is not required. A trace schema is a sequence of message-passing conditions that are expanded by the trace generators into concrete message traces, which are automatically derived from the specified properties based on the used pattern. To restrict the search-space of the trace generator, the developer can specify additional properties over the communication channels using the same specification language, which the testing procedure uses to refine the schemas derived from the specification.

We evaluated our technique by implementing it over the most popular robotic middleware, the Robot Operating System (ROS) (Quigley et al., 2009). ROS provides a communication layer that allows robotic systems to be built from components communicating through a publisher-subscriber paradigm. This instantiation comprised the implementation of the specification language for ROS messages, a test generator for trace schemas, and the deployment of runtime monitors. It was integrated into HAROS (Santos et al., 2016; Santos et al., 2021), a framework for the development of high-assurance ROS software, which automates several tasks required for analysis and reporting. Evaluation shows that schemas automatically derived from specifications are effective in finding bugs related to message interleaving, and that providing additional assumptions further improves the results.

The rest of this paper is organized as follows. Section 2 presents an overview of the proposed approach. Section 3 presents the proposed property specification language, followed by the formalization of trace schemas, their derivation and associated trace generator in Section 4. Section 5 and Section 6 present the instantiation of the approach for ROS and the evaluation results. Section 7 presents and discusses related work. Lastly, Section 8 concludes the paper.

2 SCHEMA-GUIDED TESTING

To provide an overview of the proposed framework, consider its instantiation for ROS (Section 5), used for evaluation in Section 6. ROS is a collection of libraries, tools and components for the development of robotic applications. In ROS, software is organised in *packages*, the basic build and release units. A large number of packages is open source and over 4 000 are indexed in official collections, called *distributions*. At runtime, a ROS system is typically distributed, with various independent *nodes* consuming, processing and producing data. Nodes often communicate via structured messages sent through *topics* – typed message-passing channels implementing a publisher-subscriber model.

Consider the system in Figure 2 as an example. It contains two nodes: `/planner`, which subscribes `/position` messages and publishes `/plan` messages, and `/control`, which given `/plan` and `/laser`, publishes velocity messages at `/vel` with some degree of safety. Let us say that we want to check whether a simple safety property holds in this system: that when the laser detects an obstacle closer than 40cm, a zero velocity message will be published within a certain threshold (say, 500ms), so that the robot avoids hitting the obstacle. Using the proposed specification language (Section 3), this could be easily specified as

```
globally: /laser {dist < 40} causes
  /vel {linear.x = 0} within 500ms
```

How can such a property be tested? Since nodes are considered to be black boxes, one must act at the message-passing level, publishing and listening to messages being passed by the middleware. However, not all available channels should be exercised by the testing procedure: we must identify the channels that represent the inputs of the system under test. An *open* *subscribed channel* is any channel that the system subscribes but for which there are no publishers. They are the intended interfaces to integrate the system with the environment, since there is no other purpose for subscribers that lack their respective publishers. In the example, `/position` and `/laser` are open subscriptions. Even though *closed* channels are technically usable by external components, we treat them as internal communication channels and do not tamper with them by publishing additional messages, because the system is likely not designed to expect the additional publishers. To test specific portions of the system, our approach also allows the testing of a projection of the architecture. For instance, in the example the user could wish to focus on testing the `/control` node, in which case `/plan` would become an open subscribed channel and `/position` rendered irrelevant.

Once the input channels are selected, the testing procedure must decide the channels, order and delay between the messages that will be published. Our approach is based on PBT and SBT principles. We use the former to automatically explore the input space and test the SUT with a variety of valid inputs. We use the latter to convert formal properties into specialized test strategies, in order to nudge the input generators in a direction that is more likely to reveal counterexamples.

We call these test strategies *input trace schemas*, an abstraction for a certain category of input traces. They describe a general sequence of events using abstract messages and durations, annotated with their respective constraints. For instance, for the property just presented, only input traces that publish `/laser` messages with values below 40cm are relevant, otherwise the property will never be falsified. This could be specified with the following simple trace schema.

```
        forbid /laser {dist < 40}
+0..: publish /laser {dist < 40}
```

Our system derives automatically an input trace schema for each property (Section 4.2). The notation is simple: a schema is interpreted line by line, each statement representing general constraints over channels (**`forbid`**) or mandatory events of interest (**`publish`**). The syntax follows the one used in our specification language, i.e., it expects a channel name, followed by predicates over the message's fields. The **`forbid`** statement imposes restrictions over the (zero or more) random messages an input trace is allowed to instantiate within a segment of the trace, applied from the instant immediately following the previous mandatory event (or the start of the trace) until (but not including) the next mandatory event (or until the end of the trace). In the example, the first line states that there should be no `/laser` detecting obstacles until it is forced to be published later in the trace. There is no restriction on `/position`, so any number of `/position` messages – regardless of their content – and `/laser` messages with value ≥ 40 may be published in this interval. A **`publish`** statement is composed of an interval and of constraints over a channel. It denotes a single mandatory message whose timestamp must be contained in the specified interval, relative to the previous event. In the example the derived schema does not impose any time interval, so the interval is left unbounded (0 or more milliseconds after the initial instant). After forced publication no further constraints are imposed, so any arbitrary `/position` and `/laser` messages may be published until the end of the trace.

An *input trace* is a finite sequence of messages with concrete values in data fields, annotated with timestamps generated from the schemas, that the test driver should replay on open subscribed channels to stimulate the SUT. Input traces are, thus, *instances* of a schema. Note that this is distinct from the *observed trace*, the actual sequence of events that makes a particular run of the SUT, including messages published by the nodes under test. For instance, the following input trace is an instance of the previous schema.

```
@40ms /laser    {dist: 50.0}
@50ms /position {x: 0.0, y: 0.0}
@60ms /laser    {dist: 30.0}
@70ms /position {x: 2.5, y: 0.5}
@80ms /laser    {dist: -20.0}
@90ms /position {x: 3.0, y: 1.0}
```

We can see how this trace is an instance of the previous schema: a `/laser` message with value < 40 is published after 60 milliseconds and there is no such message before it. Besides the restrictions imposed by the schema statements, the trace generator will generate arbitrary messages to build a complete input trace. However, we should also notice that it is allowing messages that are invalid in a real environment: sensors do not publish negative `/laser` messages.

To address this, the user can introduce additional assumptions following the same specification language. For the example, this could specified as:

```
globally: no /laser {dist < 0}
```

Properties that only refer to open subscriptions, called *axioms*, are considered global constraints over the input trace schemas and are used to refine them. They are used to ignore tests whose actual input traces are nonconforming. Our example schema, now refined, would be:

```
        forbid /laser {dist < 40}
+0..: publish /laser {0 <= dist < 40}
        forbid /laser {dist < 0}
```

The previously shown (invalid) input trace would no longer be generated from this schema.

Bugs often occur from specific message combinations, so the tighter the allowed message values, the higher the probability relevant input traces are generated. For instance, bugs may be triggered by the rapid publication of `/laser` messages near the 40cm boundary value. By being able to generate any positive float it would still be very unlikely to generate such a trace. Guided by specific knowledge of the SUT, the developer may introduce additional, finer axioms to force the trace generator to focus on particular classes of input traces. In our example, the developer could have imposed a very specific of range of `/laser` values, knowing that – in principle – the system would not behave differently for other values:

```
globally: no /laser {dist < 0 or
    dist > 100}
```

Such axioms should be used with care: they rule out test cases that are theoretically valid, even though they might be unexpected for a particular application.

3 SPECIFICATION LANGUAGE

The property specification language we propose is based solely on the messages that components exchange with one another in a message-oriented system. The SUT is treated as a black box, as is custom in SBT.

The language is built on top of the well-known property specification patterns proposed by (Dwyer et al., 1999). We focus on the **Absence** (a certain type of message, a *behaviour*, must not be observed), **Existence** (a certain type of message, a *behaviour*, must be observed at least once), **Precedence** (observing a certain type of message, a *behaviour*, implies that another type of message, a *stimulus*, must have been observed prior) and **Response** (observing a certain type of message, a *stimulus*, implies that another type of message, a *behaviour*, must be observed in the future) patterns. We also add a new **Prevention** pattern (a certain type of message, a *stimulus*, implies that another type of message, a *behaviour*, must *not* be observed in the future). In addition, properties specify a scope, delimiting the times during which the pattern must hold. A scope may be **Global** (the pattern should always hold), **After** (the pattern should hold indefinitely, after a certain message, the *activator*, is observed for the first time), **Until** (the pattern should hold up to the point when a certain message, the *terminator*, is observed for the first time), or **After-until** (the pattern should hold after a certain message, the *activator*, is observed, up to the point when another message, the *terminator*, is observed), the last being possibly re-entrant.

3.1 Syntax

The following grammar defines the language's syntax.

```
⟨property⟩ ::= ⟨scope⟩ ':' ⟨pattern⟩
⟨scope⟩ ::= 'globally' | ⟨after-until⟩ | ⟨until⟩
⟨after-until⟩ ::= 'after' ⟨event⟩ [⟨until⟩]
⟨until⟩ ::= 'until' ⟨event⟩
⟨pattern⟩ ::= 'no' ⟨event⟩ [⟨time-bound⟩]
  | 'some' ⟨event⟩ [⟨time-bound⟩]
  | ⟨event⟩ 'requires' ⟨event⟩ [⟨time-bound⟩]
  | ⟨event⟩ 'causes' ⟨event⟩ [⟨time-bound⟩]
  | ⟨event⟩ 'forbids' ⟨event⟩ [⟨time-bound⟩]
⟨time-bound⟩ ::= 'within' ⟨time⟩
⟨time⟩ ::= ⟨natural-zero⟩ 'ms'
⟨event⟩ ::= ⟨ros-name⟩ [⟨alias⟩] [⟨predicate⟩]
⟨alias⟩ ::= 'as' ⟨id⟩
⟨predicate⟩ ::= '{' ⟨condition⟩ '}'
⟨condition⟩ ::= ⟨value⟩ ⟨binary-rel-operator⟩ ⟨value⟩
  | '(' ⟨condition⟩ ')'
  | 'not' ⟨condition⟩
  | 'forall' ⟨id⟩ 'in' ⟨msg-field⟩ ':' ⟨condition⟩
  | ⟨condition⟩ ⟨binary-connective⟩ ⟨condition⟩
⟨binary-rel-operator⟩ ::= '=' | '<'
⟨binary-connective⟩ ::= 'or' | 'and'
⟨value⟩ ::= ⟨boolean⟩ | ⟨numeric-expr⟩ | ⟨string⟩
⟨variable⟩ ::= '$' ⟨id⟩
⟨msg-field⟩ ::= ⟨field-name⟩ {'.' ⟨field-name⟩}
  | ⟨alias-ref⟩ '.' ⟨field-name⟩ {'.' ⟨field-name⟩}
⟨alias-ref⟩ ::= '@' (⟨natural-zero⟩ | ⟨id⟩)
⟨field-name⟩ ::= ⟨id⟩ [⟨index⟩]
⟨index⟩ ::= '[' (⟨natural-zero⟩ | ⟨variable⟩) ']'
```

We present only a small set of logical connectives and relational operators, but we can assume common syntactic sugar such as the relational operator '>'.

The **`forall`** quantifier provides a basic means of iterating over all valid *indices* of array fields. The syntax '`forall i in array`' is roughly equivalent to the Python '`for i in range(len(array))`'. The grammar also captures references to quantified variables ('`$var`') and to message fields. Message fields can be: *(i)* simple identifiers ('`field`') to refer to a field of the message; *(ii)* multiple identifiers separated by dots ('`parent.child`') to refer to nested fields of composed messages; or *(iii)* indexed ('`parent.array[1]`') to access array positions.

Every message in a property has an implicit associated index, to enable cross-references with '`@`'. Messages are numbered, starting at 1 with the activator event, if any, and then following up with the pattern events. Since numeric references can become confusing, we introduce *event aliases* to allow human-readable names instead. For instance, a predicate can refer to fields of an activator '`/bumper as Bumper`' as '`@Bumper.field`' rather than '`@1.field`'. Cross-references are only possible if the referenced message (unambiguously) happens before the current message; e.g., pattern events can refer to the scope activator.

3.2 Semantics

The original specification patterns (Dwyer et al., 1999) are intended for propositional temporal logics, such as LTL, but our specification language adds first-order predicates over data and real-time constraints. Thus, we use Metric First-Order Temporal Logic (Chomicki, 1995) (MFOTL), whose syntax and semantics are similar to other commonplace logics, such as LTL, except that temporal operators in MFOTL can be annotated with metric time intervals. A temporal formula is only satisfied if it is satisfied within the bounds given by the time interval of the temporal operator, which is always relative to a timestamp.

Let Δ be the set of non-empty intervals over $\mathbb{N}_0$, such that an interval $\delta \in \Delta$ is written $[\delta_1, \delta_2)$ with $\delta_1 \in \mathbb{N}_0$, $\delta_2 \in \mathbb{N} \cup \{\infty\}$, and $\delta_1 < \delta_2$. A *signature* S is a tuple $(\mathsf{C}, \mathsf{P}, a)$, where C is a finite set of constant symbols, P is a finite set of predicates disjoint from C, and the function $a : \mathsf{P} \mapsto \mathbb{N}$ provides the arity of each predicate. Also, let V denote a countably infinite set of variables, where we assume that $\mathsf{V} \cap (\mathsf{C} \cup \mathsf{P}) = \emptyset$.

Definition 1. *The formulae over S are inductively defined: (i) For $x, y \in \mathsf{V} \cup \mathsf{C}$, $x = y$ is a formula. (ii) For $p \in \mathsf{P}$, $n = a(p)$ and $x_1, \ldots, x_n \in \mathsf{V} \cup \mathsf{C}$, $p(x_1, \ldots, x_n)$ is a formula. (iii) For $x \in \mathsf{V}$, if φ and ψ are formulae, then $(\neg\varphi)$, $(\varphi \wedge \psi)$ and $(\exists x : \varphi)$ are formulae. (iv) For $\delta \in \Delta$, if φ and ψ are formulae, then $(\bullet_\delta \varphi)$, $(\circ_\delta \varphi)$, $(\varphi \mathcal{S}_\delta \psi)$, and $(\varphi \mathcal{U}_\delta \psi)$ are formulae.*

Here $\bullet$ is read *previous*, $\circ$ *next*, $\mathcal{S}$ *since* and $\mathcal{U}$ *until*. Other common operators can be defined in terms of the core set, namely $\blacksquare$ (*historically*), $\square$ (*globally*), $\blacklozenge$ (*once*), $\lozenge$ (*finally*), $\mathcal{W}$ (*weak-until*), $\mathcal{B}$ (*back-to*) and $\mathcal{Z}$ (*weak-previous*).

A *first-order structure* D over S consists of a domain $|D| \neq \emptyset$ and interpretations $c^D \in |D|$ and $p^D \subseteq |D|^{a(p)}$, for each $c \in \mathsf{C}$ and $p \in \mathsf{P}$. A *temporal first-order structure* over S is a pair (D, τ), where $D = (D_0, D_1, \ldots)$ is a sequence of structures over S and $\tau = (\tau_0, \tau_1, \ldots)$ is a sequence of natural numbers (timestamps), where τ is monotonically increasing $(\forall i \geq 0 : \tau_i \leq \tau_{i+1})$ and makes progress $(\exists j > i : \tau_j > \tau_i)$; D has constant domains $(\forall i \geq 0 : |D_i| = |D_{i+1}|)$; and each constant symbol $c \in \mathsf{C}$ has a rigid interpretation $(\forall i \geq 0 : c^{D_i} = c^{D_{i+1}})$. A *valuation* is a mapping $v : \mathsf{V} \mapsto |D|$. We abuse notation by applying valuations also to constant symbols $c \in \mathsf{C}$, with $v(c) = c^D$. Additionally, we denote $v[x \mapsto d]$ as the valuation where every mapping is unaltered, when compared to v, save for $x \in \mathsf{V}$, which should map to $d \in |D|$.

$$\begin{array}{lll}
(D,\tau,v,i) \models x = y & \text{iff} & v(x) = v(y) \\
(D,\tau,v,i) \models x < y & \text{iff} & v(x) < v(y) \\
(D,\tau,v,i) \models p(x_1,\ldots,x_{a(p)}) & & \\
\quad \text{iff} \quad (v(x_1),\ldots,v(x_{a(p)})) \in p^{D_i} & & \\
(D,\tau,v,i) \models (\neg\varphi) & \text{iff} & (D,\tau,v,i) \not\models \varphi \\
(D,\tau,v,i) \models (\varphi \wedge \psi) & \text{iff} & (D,\tau,v,i) \models \varphi \\
 & \text{and} & (D,\tau,v,i) \models \psi \\
(D,\tau,v,i) \models (\exists x : \varphi) & \text{iff} & (D,\tau,v[x \mapsto d],i) \models \varphi, \\
 & \text{for} & \text{some } d \in |D| \\
(D,\tau,v,i) \models (\bullet_\delta \varphi) & \text{iff} & i > 0, \tau_i - \tau_{i-1} \in \delta \\
 & \text{and} & (D,\tau,v,i-1) \models \varphi \\
(D,\tau,v,i) \models (\circ_\delta \varphi) & \text{iff} & \tau_{i+1} - \tau_i \in \delta \\
 & \text{and} & (D,\tau,v,i+1) \models \varphi \\
(D,\tau,v,i) \models (\varphi \mathcal{S}_\delta \psi) & \text{iff} & (D,\tau,v,j) \models \psi \\
 & \text{and} & (D,\tau,v,k) \models \varphi \\
\quad \text{for some } j \leq i, \tau_i - \tau_j \in \delta \text{ and all } k \in [j+1, i] & & \\
(D,\tau,v,i) \models (\varphi \mathcal{U}_\delta \psi) & \text{iff} & (D,\tau,v,j) \models \psi \\
 & \text{and} & (D,\tau,v,k) \models \varphi \\
\quad \text{for some } j \geq i, \tau_j - \tau_i \in \delta \text{ and all } k \in [i, j) & &
\end{array}$$

Figure 3: Metric First-Order Temporal Logic semantics.

Definition 2. *Let (D, τ) be a temporal structure over S, with $D = (D_0, D_1, \ldots)$ and $\tau = (\tau_0, \tau_1, \ldots)$, φ and ψ formulae over S, v a valuation, and $i \in \mathbb{N}_0$. The temporal structure satisfies a formula φ, written $(D, \tau) \models \varphi$, if and only if $(D, \tau, v, 0) \models \varphi$, as shown in Figure 3.*

In the interpretation of the proposed language, the first-order structures D_i over S consist of a domain $|D|$ containing all numbers, Boolean values, strings, and messages (consisting of a unique identifier). Also, for all message channels `/ch` there is a predicate $ch \in \mathsf{P}$ such that $ch(m)$ is true if and only if a message m can be observed on `/ch`. The relation of messages to data fields is given by predicates $\mathit{field}(m, x) \in \mathsf{P}$ such that $\mathit{field}(m, x)$ holds if and only if a message m carries the value (or message) x in a field named `field`. To simplify quantification over the indices of an array, we redefine $\mathit{field}(m, k)$ to hold for all indices of the array rather than values. That is, for $k \in \mathbb{N}_0$, $\mathit{field}(m, k)$ holds if k is an index of an array named `field`. In addition, we define a predicate $\mathit{field}_k(m, x)$ that holds if the field `field[k]` carries the value x, i.e., if $\mathit{field}(m, k)$ holds and the array contains x at index k. We address composed messages, `f.g`, with the composition of predicates f and g, such that

$$g \circ f(m, x) \equiv (\exists m' : f(m, m') \wedge g(m', x))$$

Definition 3. *Let (D, τ) be a temporal structure over S, with $D = (D_0, D_1, \ldots)$ and $\tau = (\tau_0, \tau_1, \ldots)$. Let v be a valuation, $i \in \mathbb{N}_0$ and Φ a property over S. We define $[\![\Phi]\!]$, the* interpretation *of Φ, as a function that translates Φ to its equivalent MFOTL formula. Thus,*

Scopes

$$\begin{aligned}
\llbracket \texttt{globally}\colon \Psi \rrbracket &\triangleq \llbracket \Psi \rrbracket^{\emptyset}_{\bot}\\
\llbracket \texttt{until}\ q\colon \Psi \rrbracket &\triangleq \llbracket \Psi \rrbracket^{\emptyset}_{q}\\
\llbracket \texttt{after}\ p\colon \Psi \rrbracket &\triangleq \Box(\forall x : (\llbracket p \rrbracket^{x} \wedge \mathcal{Z}\,(\blacksquare(\nexists y : \llbracket p \rrbracket^{y}))) \rightarrow \llbracket \Psi \rrbracket^{x}_{\bot})\\
\llbracket \texttt{after}\ p\ \texttt{until}\ q\colon \Psi \rrbracket &\triangleq \Box(\forall x : (\llbracket p \rrbracket^{x} \wedge (\nexists y : \llbracket q \rrbracket^{y}) \wedge \mathcal{Z}((\nexists y : \llbracket p \rrbracket^{y})\ \mathcal{B}\ (\exists y : \llbracket q \rrbracket^{y}))) \rightarrow \llbracket \Psi \rrbracket^{x}_{q})
\end{aligned}$$

Patterns

$$\begin{aligned}
\llbracket \texttt{no}\ b\ \texttt{within}\ t\ \texttt{ms} \rrbracket^{\bar{x}}_{q} &\triangleq (\nexists y : \llbracket b \rrbracket^{\bar{x},y})\ \mathcal{W}_{[0,t)}\ (\exists y : \llbracket q \rrbracket^{y})\\
\llbracket \texttt{some}\ b\ \texttt{within}\ t\ \texttt{ms} \rrbracket^{\bar{x}}_{q} &\triangleq (\nexists y : \llbracket q \rrbracket^{y})\ \mathcal{U}_{[0,t)}\ ((\exists y : \llbracket b \rrbracket^{\bar{x},y}) \wedge (\nexists y : \llbracket q \rrbracket^{y}))\\
\llbracket a\ \texttt{causes}\ b\ \texttt{within}\ t\ \texttt{ms} \rrbracket^{\bar{x}}_{q} &\triangleq (\forall y : \llbracket a \rrbracket^{\bar{x},y} \rightarrow \llbracket \texttt{some}\ b\ \texttt{within}\ t\ \texttt{ms} \rrbracket^{\bar{x},y}_{q})\ \mathcal{W}\ (\exists y : \llbracket q \rrbracket^{y})\\
\llbracket a\ \texttt{forbids}\ b\ \texttt{within}\ t\ \texttt{ms} \rrbracket^{\bar{x}}_{q} &\triangleq (\forall y : \llbracket a \rrbracket^{\bar{x},y} \rightarrow \llbracket \texttt{no}\ b\ \texttt{within}\ t\ \texttt{ms} \rrbracket^{\bar{x},y}_{q})\ \mathcal{W}\ (\exists y : \llbracket q \rrbracket^{y})\\
\llbracket b\ \texttt{requires}\ a\ \texttt{within}\ t\ \texttt{ms} \rrbracket^{\bar{x}}_{q} &\triangleq \forall y : (\neg \llbracket b \rrbracket^{\bar{x},y}\ \mathcal{W}\ (\exists z : \llbracket a \rrbracket^{\bar{x},y,z} \vee \llbracket q \rrbracket^{z}))\\
&\qquad \wedge((\llbracket b \rrbracket^{\bar{x},y} \rightarrow \blacklozenge_{[0,t)}(\exists z : \llbracket a \rrbracket^{\bar{x},y,z}))\ \mathcal{W}\ (\exists z : \llbracket q \rrbracket^{z}))
\end{aligned}$$

Events

$$\begin{aligned}
\llbracket \texttt{/ch} \rrbracket^{\bar{x}} &\triangleq \llbracket \texttt{/ch}\ \{\top\} \rrbracket^{\bar{x}}\\
\llbracket \texttt{/ch}\ \{\varphi\} \rrbracket^{x_1,\dots,x_n} &\triangleq ch(x_n) \wedge \llbracket \varphi \rrbracket^{x_1,\dots,x_n}_{\emptyset}
\end{aligned}$$

Predicates

$$\begin{aligned}
\llbracket \top \rrbracket^{\bar{x}}_{\gamma} &\triangleq \top\\
\llbracket \bot \rrbracket^{\bar{x}}_{\gamma} &\triangleq \bot\\
\llbracket (\varphi) \rrbracket^{\bar{x}}_{\gamma} &\triangleq (\llbracket \varphi \rrbracket^{\bar{x}}_{\gamma})\\
\llbracket \texttt{not}\ \varphi \rrbracket^{\bar{x}}_{\gamma} &\triangleq \neg \llbracket \varphi \rrbracket^{\bar{x}}_{\gamma}\\
\llbracket \varphi\ \texttt{and}\ \psi \rrbracket^{\bar{x}}_{\gamma} &\triangleq \llbracket \varphi \rrbracket^{\bar{x}}_{\gamma} \wedge \llbracket \psi \rrbracket^{\bar{x}}_{\gamma}\\
\llbracket \varphi\ \texttt{or}\ \psi \rrbracket^{\bar{x}}_{\gamma} &\triangleq \llbracket \varphi \rrbracket^{\bar{x}}_{\gamma} \vee \llbracket \psi \rrbracket^{\bar{x}}_{\gamma}\\
\llbracket a \sim b \rrbracket^{\bar{x}}_{\gamma} &\triangleq \exists y,z : {}_{y}\llbracket a \rrbracket^{\bar{x}}_{\gamma} \wedge {}_{z}\llbracket b \rrbracket^{\bar{x}}_{\gamma} \wedge y \sim z\\
&\qquad \text{for } {\sim} \in \{=,<,\leq,>,\geq\}\\
\llbracket \texttt{forall}\ \texttt{var}\ \texttt{in}\ f\colon \varphi \rrbracket^{\bar{x}}_{\gamma} &\triangleq \forall y : {}_{y}\llbracket f \rrbracket^{\bar{x}}_{\gamma} \rightarrow \llbracket \varphi \rrbracket^{\bar{x}}_{\gamma[\texttt{var}\mapsto y]}\\
\llbracket \texttt{forall}\ \texttt{var}\ \texttt{in}\ \texttt{@}k\,.\,f\colon \varphi \rrbracket^{\bar{x}}_{\gamma} &\triangleq \forall y : {}_{y}\llbracket \texttt{@}k\,.\,f \rrbracket^{\bar{x}}_{\gamma} \rightarrow \llbracket \varphi \rrbracket^{\bar{x}}_{\gamma[\texttt{var}\mapsto y]}
\end{aligned}$$

Values and Expressions

$$\begin{aligned}
{}_{y}\llbracket c \rrbracket^{\bar{x}}_{\gamma} &\triangleq y = c && \text{for } c \text{ constant}\\
{}_{y}\llbracket \texttt{\$var} \rrbracket^{\bar{x}}_{\gamma} &\triangleq y = \gamma(\texttt{var})\\
{}_{y}\llbracket f \rrbracket^{x_1,\dots,x_n}_{\gamma} &\triangleq \llbracket f \rrbracket_{\gamma}(x_n, y)\\
{}_{y}\llbracket \texttt{@}k\,.\,f \rrbracket^{x_1,\dots,x_n}_{\gamma} &\triangleq \llbracket f \rrbracket_{\gamma}(x_k, y) && \text{if } 1 \leq k \leq n\\
{}_{y}\llbracket (a) \rrbracket^{\bar{x}}_{\gamma} &\triangleq ({}_{y}\llbracket a \rrbracket^{\bar{x}}_{\gamma})\\
{}_{y}\llbracket a \oplus b \rrbracket^{\bar{x}}_{\gamma} &\triangleq \exists z,w : {}_{z}\llbracket a \rrbracket^{\bar{x}}_{\gamma} \wedge {}_{w}\llbracket b \rrbracket^{\bar{x}}_{\gamma} \wedge y = (z \oplus w)\\
&\qquad \text{for } \oplus \in \{+,-,\times,\div\}
\end{aligned}$$

Field Predicates

$$\begin{aligned}
\llbracket \texttt{field} \rrbracket_{\gamma} &\triangleq \mathit{field}\\
\llbracket \texttt{field[}k\texttt{]} \rrbracket_{\gamma} &\triangleq \mathit{field}_{k}\\
\llbracket \texttt{field[\$var]} \rrbracket_{\gamma} &\triangleq \mathit{field}_{\gamma(\texttt{var})}\\
\llbracket f_1\,.\,f_2 \rrbracket_{\gamma} &\triangleq \llbracket f_2 \rrbracket_{\gamma} \circ \llbracket f_1 \rrbracket_{\gamma}
\end{aligned}$$

Figure 4: Specification language semantics.

we define $(D,\tau,v,i) \vDash \Phi$ *as* $(D,\tau,v,i) \vDash \llbracket \Phi \rrbracket$*, and* $\llbracket \Phi \rrbracket$ *is defined as shown in Figure 4.*

To shorten some formulae and improve readability, we assume that all interpreted properties Φ are type-checked prior to their semantic interpretation $\llbracket \Phi \rrbracket$.

Scopes. `globally` and `until` start, by definition, at the initial instant of the trace and, thus, require the inner formula to hold at that instant. On the other hand, `after` and `after-until` start when an activator is been observed – hence the $\Box(\mathit{activator} \rightarrow \Psi)$ type of formula. The translation of a pattern Ψ is denoted by $\llbracket \Psi \rrbracket^{\bar{x}}_{q}$ and is passed a vector of quantified messages $\bar{x} = x_1,\dots,x_n$ – initially the activator, if any – and the terminator event q, if any.

Patterns. We present only the timed variants of each pattern, since untimed variants are particular cases where $\delta = \infty$. Unary patterns, `no` and `some`, are already close to a MFOTL formula, requiring the translation of event predicates, denoted by $\llbracket a \rrbracket^{\bar{x}}$ for an event a given quantified messages $\bar{x}$. The binary patterns `causes` and `forbids` are translated via composition with the unary ones. The most complex pattern, `requires`, is defined with two conjuncts: behaviour b should not happen before its stimulus a or the scope terminator q; and if b happens before q, then a must have happened within the previous δ time instants.

Predicates. The translation of a predicate φ is mostly direct to logic values or connectives, and is denoted by $\llbracket \varphi \rrbracket^{\bar{x}}_{\gamma}$ for quantified messages $\bar{x}$ and a mapping γ for syntactic replacements arising from quantifications (given a variable name *as written in the property*, its occurrences are replaced with a quantified variable in $|D|$). When translating an expression a, ${}_{y}\llbracket a \rrbracket^{\bar{x}}_{\gamma}$ denotes a predicate that tests whether the value of the variable y is one of the values for the expression a.

Values and Expressions. Message fields translate to their homonymous predicates, and field composition to predicate composition, as previously explained. References to other messages, e.g., `@`$k.f$, convert the human-readable aliases to the message index, referring to the k-th message of the superscript vector, x_k, rather than the current message, x_n. Lastly, arithmetic expressions bind quantified variables to domain values.

4 TRACE SCHEMAS

This section formalizes the core notion of the proposed approach, that of input trace schemas. Then it shows how they integrate the approach by presenting how schemas are derived from patterns, and then concrete input traces generated from schemas.

4.1 Formalization

A trace schema is a sequence of restrictions on a sequence of messages, forcing messages with certain properties to be published and forbidding certain other kinds of messages between such publications.

Definition 4. *An* input trace schema Γ *is a set of formulae over a signature S, written as a sequence of statements according to the following grammar.*

$$\begin{array}{l}
\langle schema \rangle ::= \ \langle statement \rangle^* \\
\langle statement \rangle ::= \text{`}\mathtt{forbid}\text{'}\ \langle event \rangle \\
\quad |\quad \langle bounds \rangle\ \text{`}\mathtt{:}\text{'}\ \text{`}\mathtt{publish}\text{'}\ \langle event \rangle \\
\langle bounds \rangle ::= \ \text{`}\mathtt{+}\text{'}\ \langle natural\text{-}zero \rangle\ \text{`}\mathtt{..}\text{'}\ [\langle natural\text{-}zero \rangle]
\end{array}$$

To convert schemas into MFOTL formulae, let us start by defining an atomic variable e_0 for the initial instant of the trace and atomic variables e_i for each of the k **`publish`** events in a schema. Each atomic variable e_i is true only at the instant when the i^{th} **`publish`** event happens. Given statements of the form

$$+m_i\texttt{..}n_i\texttt{:}\ \textbf{publish}\ \texttt{/ch}\ \{\varphi_i\}$$

for all $i \in \mathbb{N}$, we have that: $\Box(e_i \to \circ(\Box\neg e_i))$, i.e., once e_i happens, it will not happen again; and also $\Box(e_i \to (\exists x : ch(x) \wedge \varphi_i))$, i.e., at the instant e_i happens, there is a message on `/ch` such that φ_i. Then, we must schedule all mandatory events. For all $i \leq k$ we must observe $\Box(e_{i-1} \to \Diamond_{[m_i,n_i)} e_i)$. Then, the constraints given by **`forbid`** statements must be considered. Given a statement **`forbid`** `/ch` $\{\varphi\}$ placed between events e_i and e_{i+1}, it follows that: $\Box(e_i \to \circ((\neg\exists x : ch(x) \wedge \varphi)\ \mathcal{W}\ e_{i+1}))$. The $\circ$ and $\mathcal{W}$ operators ensure that the constraints do not apply at the logical instants e_i and e_{i+1}. If the schema ends with a **`forbid`** statement, the $\mathcal{W}$ operator will impose the constraints indefinitely, as $e_{k+1} \equiv \bot$.

$$\begin{array}{lcl}
\langle \textbf{globally}\texttt{:}\ \Psi \rangle & \triangleq & \langle \Psi \rangle \\
\langle \textbf{until}\ q\texttt{:}\ \Psi \rangle & \triangleq & \langle \Psi \rangle \\
\begin{array}{l}\langle \textbf{after}\ p\texttt{:}\ \Psi \rangle \\ \langle \textbf{after}\ p\ \textbf{until}\ q\texttt{:}\ \Psi \rangle\end{array} & \triangleq & \boxed{\begin{array}{rl} & \textbf{forbid}\ p \\ \texttt{+0..:} & \textbf{publish}\ p \\ \multicolumn{2}{l}{\langle \Psi \rangle}\end{array}} \\
\langle \textbf{no}\ b\ \textbf{within}\ t\ \textbf{ms} \rangle & \triangleq & \varepsilon \\
\langle \textbf{some}\ b\ \textbf{within}\ t\ \textbf{ms} \rangle & \triangleq & \varepsilon \\
\langle b\ \textbf{requires}\ a\ \textbf{within}\ t\ \textbf{ms} \rangle & \triangleq & \varepsilon \\
\begin{array}{l}\langle a\ \textbf{causes}\ b\ \textbf{within}\ t\ \textbf{ms} \rangle \\ \langle a\ \textbf{forbids}\ b\ \textbf{within}\ t\ \textbf{ms} \rangle\end{array} & \triangleq & \boxed{\begin{array}{rl} & \textbf{forbid}\ a \\ \texttt{+0..:} & \textbf{publish}\ a\end{array}}
\end{array}$$

Figure 5: Translation from specifications to schemas.

4.2 Generating Input Traces

From Patterns to Schemas. For each pattern of the property language we specify how a *base schema* is derived. These schemas represent a set of traces with minimal constraints such that every counterexample to the property is necessarily in them (i.e., they are complete). These schemas essentially take into consideration the activators and stimuli of the properties, without which no counterexample can be found.

This translation between properties and base schemas is shown in Figure 5, and is mostly self-explanatory, denoted by $\langle\Phi\rangle$ for a property Φ. For example, consider the base schema for a **`causes`** b: a stimulus is forced to be published, otherwise the property cannot be falsified.

Refining Schemas. Axioms play an important role in establishing valid data and valid trace structure. In some cases, they can be embedded into the base schemas, so that generated traces are mostly correct by construction. The refinement of a schema Γ by an axiom Φ is denoted by $\lfloor\Gamma\rfloor_\Phi$. We support refinements for two types of properties, specifically:

```
globally: no /c {φ}
globally: /c forbids /c within t ms
```

If an axiom Φ is of the first shape, constraints are added to the data a message can carry. For `/c` $\neq$ `/d`:

$$\left\lfloor \begin{array}{l} \Gamma_1 \\ \delta\texttt{:}\ \textbf{publish}\ \texttt{/d}\ \{\psi\} \\ \Gamma_2 \end{array} \right\rfloor_\Phi \triangleq \boxed{\begin{array}{l} \Gamma_1 \\ \quad \textbf{forbid}\ \texttt{/c}\ \{\varphi\} \\ \delta\texttt{:}\ \textbf{publish}\ \texttt{/d}\ \{\psi\} \\ \quad \textbf{forbid}\ \texttt{/c}\ \{\varphi\} \\ \Gamma_2 \end{array}}$$

Publish statements on the same channel `/c` also have their predicates restricted:

$$\left[\begin{array}{l} \Gamma_1 \\ \delta\text{: } \texttt{publish /c } \{\psi\} \\ \Gamma_2 \end{array}\right]_\Phi \triangleq \boxed{\begin{array}{l} \Gamma_1 \\ \quad \texttt{forbid /c } \{\varphi\} \\ \delta\text{: } \texttt{publish /c} \\ \qquad \{\psi \texttt{ and not } \phi\} \\ \quad \texttt{forbid /c } \{\varphi\} \\ \Gamma_2 \end{array}}$$

Axioms Φ of the second shape impose additional temporal constraints. For two consecutive **publish** statements on channel /c, we shift the lower-bound of the second interval, if $\texttt{t} < \texttt{n}$:

$$\left[\begin{array}{l} \Gamma_1 \\ \delta\text{: } \texttt{publish /c } \{\psi\} \\ \Gamma_2 \\ \texttt{m..n:} \\ \quad \texttt{publish /c } \{\psi'\} \\ \Gamma_3 \end{array}\right]_\Phi \triangleq \boxed{\begin{array}{l} \Gamma_1 \\ \delta\text{: } \texttt{publish /c } \{\psi\} \\ \Gamma_2 \\ \texttt{t..n:} \\ \quad \texttt{publish /c } \{\psi'\} \\ \Gamma_3 \end{array}}$$

If $\texttt{t} \geq \texttt{n}$, the specification is a contradiction.

From Schemas to Traces. The last step is to generate traces that conform to the schemas.

Definition 5. *Let S be a signature, v a valuation and Γ an input trace schema over S. A temporal first-order structure (D, τ) over S is said to be an* instance *of Γ if and only if, for all $\varphi \in \Gamma$, it is true that $(D, \tau, v, 0) \vDash \varphi$.*

This definition determines whether a temporal first-order structure (an infinite trace) is an instance of a schema, but we can only generate finite input traces. Thus, we need to validate finite traces against schemas.

Definition 6. *An* input trace *of length k is a k-prefix of a temporal structure over S, for some $k \in \mathbb{N}_0$.*

Definition 7. *Let S be a signature, v be a valuation and Γ be an input trace schema over S. Let $k \in \mathbb{N}_0$ and $(D_{[0,k)}, \tau_{[0,k)})$ be a k-prefix of a temporal structure over S. The k-prefix $(D_{[0,k)}, \tau_{[0,k)})$ is said to be an* instance *of Γ if and only if, for all (D', τ') temporal structures over S, $(D_{[0,k)}, \tau_{[0,k)})$ is a k-prefix of (D', τ'), and it is true that $(D', \tau', v, 0) \vDash \varphi$, for all $\varphi \in \Gamma$.*

The previous definition, in essence, states that, for a finite trace to be an instance of a schema, any of its possible (infinite) extensions must also be instances of the same schema, as is standard in bounded temporal semantics. With this, we are now equipped to produce traces that comply with a set of constraints.

The algorithm for trace generation is simple. An initial pass allocates all timestamps and messages associated with **publish** events. A second pass iterates over all intervals before and after the mandatory events, and, evaluating the applicable **forbid** statements for each interval, builds a set of all usable channels. From this set, it then produces a random number of messages, and intersperses the messages randomly along the interval. The result is a sorted list of *(timestamp, channel, message)* tuples; a finite trace.

5 TOOL SUPPORT FOR ROS

Given a system specification, composed of behavioural properties following the syntax shown in Section 3 and a description of the inputs and outputs of the SUT, the proposed schema-based testing approach automatically generates property-based tests that try to falsify the specification. Our implementation is compositional, tackling one problem at a time, namely: property parsing[1], runtime monitor generation[2] and data generation for trace schemas[3]. To obtain the necessary architecture model of the SUT, we implemented the test generator as a plug-in for the HAROS framework, which is already capable of extracting models from source code (Santos et al., 2019). This eases workflow automation, from model extraction to test generation, execution and reporting.

From a high-level perspective, the tool starts by filtering the provided properties. Each property is assigned one category of *axiom*, *testable* or *non-testable*. Although in theory all properties can be tested, our current implementation for ROS only considers a property testable if we have complete control of its scope and stimulus events. That is, its scope activator, scope terminator and stimulus event (if applicable) must be open subscribed topics (inputs), while its behaviour event must be a published topic (output). Furthermore, predicates over input messages must not contain arithmetic or quantified expressions. These limitations help ensure that tests are feasible and reproducible. Predicates over input messages are embedded into the message generators on a best-effort basis, which still requires a subsequent runtime validation step. Quantifiers and arithmetic expressions make constraints over data significantly more complex, meaning that more input candidates have to generated until one is valid. Limiting outputs only to behaviour events grants the test driver control over the pace of the test; the input trace schema dictates when scopes should start and when behaviours should be triggered.

For each testable property, the tool identifies the property's pattern and produces the corresponding base schemas, as presented in Section 4. Axioms are considered next, if applicable, to refine all starting schemas. Refinement is a best-effort attempt at generating correct data by construction. Again, it does not capture all given constraints and must be validated at runtime.

Following the schema generation and refinement steps, the tool applies meta-programming Jinja[4] templates for all properties, to produce source code blocks

[1]https://github.com/git-afsantos/hpl-specs
[2]https://github.com/git-afsantos/hpl-rv-gen
[3]https://github.com/git-afsantos/haros-plugin-pbt-gen
[4]https://palletsprojects.com/p/jinja/

for the runtime monitors, trace generators and the overall test case structure. Each property pattern has its corresponding code template, meaning that we can produce optimized monitors and data generators for our limited catalogue of options. The trace generators are implemented as data generators for Hypothesis (MacIver et al., 2019), a well-established PBT library. At the end, all pieces are weaved into a single test script, which is responsible for successively producing traces using Hypothesis, according to a given schema, launching the SUT using tools of the ROS infrastructure, deploying runtime monitors, replaying the input trace and then observing the results. The test script is itself a ROS node, allowing runtime monitors to subscribe directly to all topics in the system, and, thus, observe messages as they are published.

6 EVALUATION

The ROS-based implementation of the proposed schema-based approach for PBT has been applied to two concrete robotics applications. Note that the SUT are treated as black boxes with a publisher-subscriber interface, which makes the actual number of ROS nodes in the network irrelevant. Thus, for both case studies, we tested both individual nodes and fully integrated systems. We present now our case studies, the methodology behind our evaluation of the approach and, lastly, the observed results.

6.1 Case Studies

The iClebo Kobuki[5] is our first test subject and one of the most iconic ROS robots. It is a low-cost, personal robot kit with open-source software whose main purpose is to provide an entry-level platform for roboticists to build applications with. Kobuki provides for an interesting case study, since its source code has been relatively well maintained and there is online documentation easily available – which makes it easier to understand the expected behaviour of the system.

AgRob V16[6] is a modular robotic platform for hillside agriculture, adapted to steep slope vineyards. There are a few appealing factors in AgRob V16 as a case study, namely that it is an industrial robot, more complex than Kobuki, and that a large part of its software comes from integrated third-party packages.

For this evaluation, and for both robots, we consider a configuration with three components: a Trajectory Controller node, that provides velocity commands to the robot; a Safety Controller node, responsible for reacting to dangerous sensor readings; and a Supervisor node, that selects which commands to pass down to the hardware, based on the current safety state and the priority of the command source.

6.2 Methodology

One of the characteristics of PBT is that it can only find discrepancies between implementation and specification (Hughes, 2016). Whether such discrepancies are an error in the system or in the specification is left to the user to decide. We have spent months studying the implementations of our target systems. Judging from the available documentation, we assume that they are relatively stable, with correct functional behaviour for the most part. This is a crucial assumption, as we write specifications mostly based on the actual (and expected) behaviour of the implementation. Given that PBT focuses on discrepancies, our evaluation process involves two types of testing:

Positive Testing is the act of testing properties that we know to be true. The goal is to confirm that the tool does not introduce false positives – i.e., that it does not report an error if there is none to report.

Negative Testing is the act of testing properties that we know to be false. The tool should be able to find at least one counterexample, athough this may require tuning how many examples are tested.

We follow a systematic method to write specifications that include both true and false properties. We start by writing a catalogue of true properties and axioms for each system. Then, we apply *specification mutation* (Budd and Gopal, 1985; Ammann and Black, 1999; Black et al., 2000; Trab et al., 2012) to construct *mutants* (i.e., variants) of the initial set. Mutations are often small changes, such as changing a single operator or a variable. If the initial properties are as precise as we can write them, any small mutation should end up generating a false property. In practice, a validation step is still required to discard duplicates, trivial properties and *equivalent mutants* (mutants that are still true properties). Mutant validation is done manually, since it is an undecidable problem (Trab et al., 2012).

The existing literature provides many operators to mutate specifications. We adapt and reuse those that are applicable to our specification language, such as: replacing an **operand** in an expression with another valid operand of the same type (e.g., `1` with `0`); **negating** an expression; replacing a **logical operator** (e.g., `and` with `or`); replacing a **relational operator** (e.g., $<$ with $>$); **removing a predicate** from an event (e.g., `/p {φ}` becomes `/p`); widening or shortening

[5]http://kobuki.yujinrobot.com/about2/

[6]http://agrob.inesctec.pt/

time constraints; replacing a **message channel** with another of the same type; widening the **scope** (e.g., `after /p {φ}` becomes `globally`); and replacing the property's **pattern** (e.g., `causes` with `forbids`).

After building the final catalogue of property mutants, we run the PBT tool on each mutated property multiple times, with different parameters. With this, we aim to answer the following research questions: **RQ1**, how effective are the base schemas we introduce in Section 4, when compared to a pure random approach; **RQ2**, how effective is a specification with stronger axioms; and **RQ3**, what efficiency gains can we make from base schemas and stronger axioms.

To answer these questions, we ran the tool with *empty schemas* (effectively turning tests into a naïve PBT approach with random exploration); with the automatically derived *base schemas*; and with both empty and base schemas refined through *stronger axioms*. To evaluate RQ1 and RQ2, we compare the resulting *specification coverage* (Ammann and Black, 1999), a metric that is, in essence, a *mutation score*, i.e., the ratio of mutants that the testing suite kills. For RQ3, for each property mutant we measured the number of examples until a test fails for the first time; the number of examples until the final test result (includes the PBT shrinking phase); and the required time to run each example (in full, including data generation, launching the SUT, and replaying the example trace).

6.3 Results

The evaluation results we present here are an aggregated overview of all SUTs and test runs (6 individual nodes and 2 full systems). In total, our specification is comprised of 153 axioms, 33 true properties and 318 mutants. Some axioms from testing single nodes are replicated to the integrated system, making the total number of unique axioms 82.

Effectiveness. Regarding the positive testing step, no false positives have been observed in any of the SUTs, resulting in a precision of 1. As for the negative testing step, for the total of 318 mutants, and an initial limit of 200 examples per property, our schema-based approach achieved a mutant score (or recall) of 0.909 (29 surviving mutants), while the naïve approach only achieved a score of 0.808 (62 survivors). All mutants that survived the base schema also survived the empty schema. Increasing the limit to 500 examples per property did not make a difference for the empty schemas, but dropped the survivors to 27 with the base schemas. These surviving mutants are all very similar, belonging to 5 properties of AgRob V16 (one of them being the root of 13 false negatives). The expected counterexamples for these mutants require sequences of 3 or more messages, with both timing and content constraints. Answering RQ1, the base schemas represent an increase in recall of 13.24% (from 0.808 to 0.915).

Next, we reverted the testing limit to 200 examples and strengthened 16 axioms affecting the surviving mutants. The mutant score increased to 0.934 (21 surviving mutants) with our base schemas – a better improvement than simply running more examples. The empty schema also achieved an increased mutant score, of 0.858 (45 survivors). Again, raising the testing limit to 500 examples was not enough for the naïve approach, leaving 35 survivors (score of 0.89). For our base schemas, however, the increased limit was sufficient to kill all mutants. Answering RQ2, stronger axioms represented a boost in recall as high as 12.35%.

Efficiency. On average, each example takes about 8 seconds to set up and execute, including generating the trace, and starting and shutting down the SUT. However, this number tends to decrease the less examples we run, as the PBT library requires less effort to find new inputs. All properties and mutants combined required a total of 43 029 examples until our base schemas killed all mutants, which makes for a total runtime of about 98 hours and 18 minutes. Considering only the cases for which the tool found a counterexample within 200 examples, it took, on average, 6 examples (52 seconds) to find an initial counterexample and 27 examples to report the final counterexample (226 seconds). This difference is due to the example shrinking phase of PBT, which spanned nearly half of the negative testing runtime (17 959 examples out of 36 429). While the average time per example is similar for an empty schema, the total number of examples tried for the whole test suite jumps up to 92 800, for a total runtime of about 211 hours and 59 minutes. A pure random approach takes longer not only to find an initial counterexample (12 examples), but also to shrink it and report the final counterexample (58 examples). Overall, and answering RQ3, we observe that the schema-based approach is between 3 times (median) and 7 times (average) faster than the random approach in all measures, if there is a counterexample to be found. In addition, for a limit of 500 examples, stronger axioms killed an additional 27 mutants with our approach, which amounts to sparing 5 697 examples, or about 12 hours and 40 minutes of runtime.

7 RELATED WORK

Specification-based Testing. Most of the existing literature on SBT shares a few common traits. First, regarding the tested specifications, there is a general lack of high-level (possibly pattern-based) specification languages. Properties are encoded directly in formal logics, often in LTL, which makes our approach more easily usable by non-experts. Second, the focus on LTL means that specifications cannot impose constraints related to real-time performance, as we can with MFOTL semantics. Third, few approaches target publisher-subscriber architectures over (possibly) distributed systems. A notable exception is the work of Michlmayr et al., which does target publisher-subscriber systems, although these are non-distributed Java applications (Michlmayr et al., 2006).

Different strategies and techniques are used to explore relevant test cases from LTL specifications. In the case of Michlmayr et al., it is unclear whether there is a concrete strategy behind input generation. Tan et al. convert the original formula into a finite set of formulae characterizing non-trivial tests (Tan et al., 2004); these are then fed to a model checker, whose counterexamples are turned into concrete inputs. Arcaini et al., and later Narizzano et al., build online monitors based on Büchi automata from LTL specifications over sequences of function calls (Arcaini et al., 2013; Narizzano et al., 2020). The monitors serve both as a test oracle and as an automaton whose states are used to generate test cases, according to various coverage criteria. The former focuses on safety properties only, while Narizzano et al. consider also bounded liveness (co-safety) properties. Bloem et al. follow a similar approach, but deem requirements to be too abstract to fully predict the behaviour of a black-box system (Bloem et al., 2019). Their online testing approach asks the user for a system specification and a fault model defining coverage goals (rather than a behaviour model), both specified in LTL. Test cases are synthesized from an automata of the fault model.

Property-based Testing. Work on PBT often acts at a much lower level of abstraction than we propose in this paper. The notion of system specification is lacking; properties are encoded programmatically and tend to focus on simpler assertions, rather than temporal logic formulae. A partial exception to this pattern is FlinkCheck (Espinosa et al., 2019), a PBT tool for Apache Flink, a stream processing framework. Properties over streams are defined programmatically, but formalized in LTL_{ss}, a three-valued variant of LTL with timeouts on operators.

One of the major problems tackled in PBT research is input space exploration – how data is generated. Pure random generation is generally considered inefficient, which led to novel techniques to guide input generators. ConFuzz (Padhiyar and Sivaramakrishnan, 2021) is a PBT technique for event-driven programs that uses a state-of-the-art grey-box fuzzer to target concurrency bugs. It focuses on event callback scheduling, not on the notion of traces of events. Targeted PBT is an enhanced form of PBT where input generation is guided by a search strategy and a utility function (Löscher and Sagonas, 2017). The latter attributes a score to each input-output pair that determines how close an input was to falsify the property, which should be maximized. Coverage-guided PBT (Padhye et al., 2019) is another mechanism to guide test generation, here using coverage information provided by the execution of the previous tests. In our case, it is the structure of the high-level specification that enables us to guide input generation and discard a large portion of irrelevant message traces.

Lastly, various authors have been researching the potential of PBT and fuzzing in the context of ROS systems (Santos et al., 2018; Woodlief et al., 2021; Delgado et al., 2021). These works confirm the bug detection effectiveness of these techniques in complex robotic systems, but they lack high-level property specifications. They provide mostly message generators, with limited degrees of input generation guidance.

8 CONCLUSION

Testing distributed message-passing systems is a complex task, for there are many possible causes of failure, such as message interleaving, message contents or timing. In this paper we presented a novel schema-guided testing approach for such systems that combines principles from both Specification-based Testing and Property-based Testing. With the former, we are able to identify classes of relevant inputs for a given temporal property, here called schemas. With the latter, we have an automated mechanism to instantiate schemas into actual traces of messages to replay to the system under test. Specifications are composed of both assumptions and properties to be tested, formalized with Metric First-Order Temporal Logic to support real-time constraints and constraints over complex data structures. Contrary to most existing approaches, however, specifications are written using a high-level, pattern-based specification language, also presented in this paper, to ease the learning curve.

We have applied the proposed approach to robotic systems implemented with the popular Robot Operating System middleware. Our results show that schema-guided testing is both more effective and more efficient than a pure PBT approach, even when using minimal schemas. Our first goal for future research is to explore whether we could further boost performance by partitioning minimal schemas into several specialized schemas that, as a whole, are equivalent to the current schemas. Second, our approach to automatic schema refinement via user-provided assumptions can be improved. Many assumptions are only validated at runtime in the current version, which wastes time in data generation. Third, we will open up the implementation to accept arbitrary, user-provided schemas, in addition to properties. Finally, we also envision improvements to the specification language, so as to support other property patterns or full-blown state machines.

ACKNOWLEDGEMENTS

The research leading to these results has received funding support from the projects: ERDF - European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme; project PTDC/CCI-INF/29583/2017 (POCI-01-0145-FEDER-029583) financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia; "STEROID - Verification and Validation of ADAS Components for Intelligent Vehicles of the Future" from the European Union Financial Support (FEDER) under grant agreement No. 69989; and "NORTE-06-3559-FSE-000046 - Emprego altamente qualificado nas empresas – Contratação de Recursos Humanos Altamente Qualificados (PME ou CoLAB)" financed by the Norte's Regional Operational Programme (NORTE 2020) through the European Social Fund (ESF).

REFERENCES

Ammann, P. and Black, P. E. (1999). A specification-based coverage metric to evaluate test sets. In *HASE*, pages 239–248. IEEE CS.

Arcaini, P., Gargantini, A., and Riccobene, E. (2013). Online testing of LTL properties for java code. In *Haifa Verification Conference*, volume 8244 of *LNCS*, pages 95–111. Springer.

Black, P. E., Okun, V., and Yesha, Y. (2000). Mutation operators for specifications. In *ASE*, pages 81–88. IEEE CS.

Bloem, R., Fey, G., Greif, F., Könighofer, R., Pill, I., Riener, H., and Röck, F. (2019). Synthesizing adaptive test strategies from temporal logic specifications. *Formal Methods Syst. Des.*, 55(2):103–135.

Budd, T. A. and Gopal, A. S. (1985). Program testing by specification mutation. *Computer Languages*, 10(1):63–73.

Chomicki, J. (1995). Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.*, 20(2):149–186.

Delgado, R., Campusano, M., and Bergel, A. (2021). Fuzz testing in behavior-based robotics. In *ICRA*, pages 9375–9381. IEEE.

Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1999). Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420. ACM.

Espinosa, C. V., Martin-Martin, E., Riesco, A., and Rodríguez-Hortalá, J. (2019). FlinkCheck: Property-based testing for Apache Flink. *IEEE Access*, 7:150369–150382.

Hughes, J. (2016). Experiences with QuickCheck: Testing the hard stuff and staying sane. In *A List of Successes That Can Change the World*, volume 9600 of *LNCS*, pages 169–186. Springer.

Löscher, A. and Sagonas, K. (2017). Targeted property-based testing. In *ISSTA*, pages 46–56. ACM.

MacIver, D., Hatfield-Dodds, Z., and Contributors, M. (2019). Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891.

Michlmayr, A., Fenkam, P., and Dustdar, S. (2006). Specification-based unit testing of publish/subscribe applications. In *ICDCS Workshops*, page 34. IEEE CS.

Narizzano, M., Pulina, L., Tacchella, A., and Vuotto, S. (2020). Automated requirements-based testing of black-box reactive systems. In *NFM*, volume 12229 of *LNCS*, pages 153–169. Springer.

OMG (2015). *Data Distribution Service (DDS), Version 1.4*. Object Management Group (OMG).

Padhiyar, S. and Sivaramakrishnan, K. C. (2021). ConFuzz: Coverage-guided property fuzzing for event-driven programs. In *PADL*, volume 12548 of *LNCS*, pages 127–144. Springer.

Padhye, R., Lemieux, C., and Sen, K. (2019). JQF: coverage-guided property-based testing in java. In *ISSTA*, pages 398–401. ACM.

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). ROS: An open-source Robot Operating System. In *ICRA Workshop on Open Source Software*.

Santos, A., Cunha, A., and Macedo, N. (2018). Property-based testing for the robot operating system. In *A-TEST@ESEC/SIGSOFT FSE*, pages 56–62. ACM.

Santos, A., Cunha, A., and Macedo, N. (2019). Static-time extraction and analysis of the ROS computation graph. In *IRC*, pages 62–69. IEEE.

Santos, A., Cunha, A., and Macedo, N. (2021). The high-assurance ROS framework. In *RoSE@ICSE*, pages 37–40. IEEE.

Santos, A., Cunha, A., Macedo, N., and Lourenço, C. (2016). A framework for quality assessment of ROS repositories. In *IROS*, pages 4491–4496. IEEE.

Tan, L., Sokolsky, O., and Lee, I. (2004). Specification-based testing with linear temporal logic. In *IRI*, pages 493–498. IEEE SMCS.

Trab, M. S. A., Counsell, S., and Hierons, R. M. (2012). Specification mutation analysis for validating timed testing approaches based on timed automata. In *COMPSAC*, pages 660–669. IEEE CS.

Woodlief, T., Elbaum, S., and Sullivan, K. (2021). Fuzzing mobile robot environments for fast automated crash detection. In *ICRA*, pages 5417–5423. IEEE.